UNIVERSITAT POLITECNICA DE CATALUNYA

PARALLELISM

Lab 5: Geometric (data) decomposition: heat diffusion equation

Roger Vilaseca Darne and Xavier Martin Ballesteros
PAR4110

7th June 2019, Q2

Contents

1 Introduction

2 Analysis with *Tareador*

In this section we have used the *Tareador* tool to analyse the possible parallelism strategies that we can use in order to parallelise both Jacobi and Gauss-Seidel solvers.

We have mainly focused on the data dependences that appear and how will we protect them in our parallel *OpenMP* code. To explore the dependences, we have used a much finer task decomposition: one task for each iteration of the body of the most innerloop.

2.1 Jacobi Solver

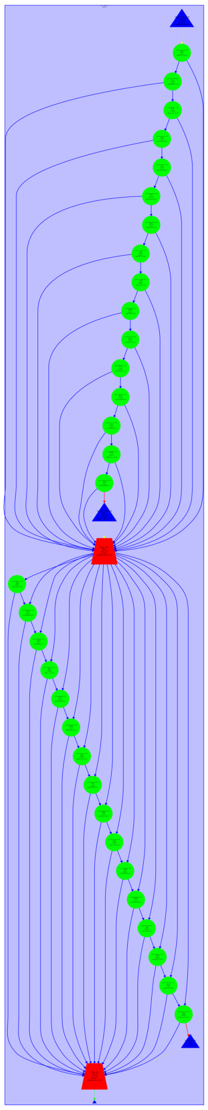
This solver uses an auxiliar matrix **utmp** to write the resulting values of the computation in the most innerloop of the body. For each element in the matrix **u**, it takes the values of the element on its top and bottom and on its left and right, and does some arithmetic operations.

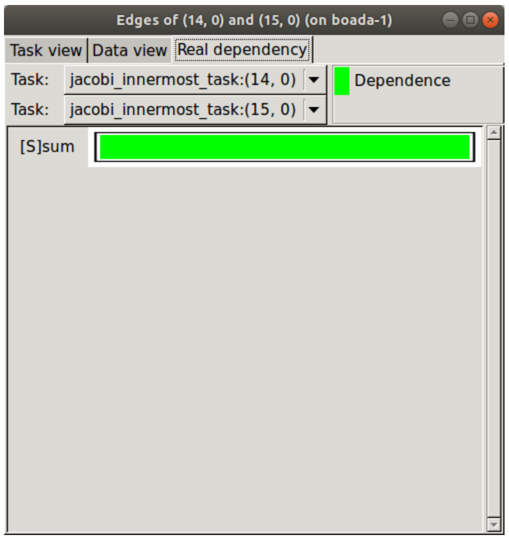
The modified code using the *Tareador* tool is shown below. However, the code can be found in *jacobi-tareador.c* file inside the codes directory.

```c
double relax_jacobi (double *u, double *utmp, unsigned sizex,
                     unsigned sizey)
{
    double diff, sum=0.0;

    int howmany=1;
    for (int blockid = 0; blockid < howmany; ++blockid) {
      int i_start = lowerb(blockid, howmany, sizex);
      int i_end = upperb(blockid, howmany, sizex);
      for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
        for (int j=1; j<= sizey-2; j++) {
          tareador_start_task("jacobi_innermost_task");
          utmp[i*sizey+j]= 0.25 * ( u[ i*sizey     + (j-1) ]+   // left
                                    u[ i*sizey     + (j+1) ]+   // right
                                    u[ (i-1)*sizey + j     ]+   // top
                                    u[ (i+1)*sizey + j     ]); // bottom
          diff = utmp[i*sizey+j] - u[i*sizey + j];
          sum += diff * diff;
          tareador_end_task("jacobi_innermost_task");
        }
      }
    }
    return sum;
}
```

Figure 1: Code for the task decomposition for relax_jacobi function.

With this modified version of the code we can obtain the task decomposition graph (TDG), which can be found in Figure 2a. Moreover we can see that there exist some kind of data dependencies between tasks. To know which variable is the responsible of these dependences, we have right clicked in an edge between two jacobi_innermost_task nodes (green) >> Dataview >> Edge >> Real dependency. The obtained results are shown in Figure 2b.


(a)


Edges of (14, 0) and (15, 0) (on boada-1)

Task view | Data view | Real dependency

Task: jacobi_innermost_task:(14, 0)

Task: jacobi_innermost_task:(15, 0)

🟩 Dependence

[S]sum 🟩

(b)

Figure 2: (a) Task decomposition graph for the Jacobi solver, (b) Real data dependences in the Jacobi solver.

Now, we know that the **sum** variable creates the dependences for the Jacobi solver. Novertheless, we have made use of some *Tareador* calls that temporarily filter the analysis for the variable sum, causing the serialization and obtaining a new task graph (Figure **??**). The modified fragment of the code can be found in *jacobi-tareador-disable-sum.c* file in the codes directory.

```
double relax_jacobi (double *u, double *utmp, unsigned sizex,
                     unsigned sizey)
{
    ...
    for (int blockid = 0; blockid < howmany; ++blockid) {
      ...
      for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
        for (int j=1; j<= sizey-2; j++) {
            tareador_start_task("jacobi_innermost_task");
            ...
            tareador_disable_object(&sum);
            sum += diff * diff;
            tareador_enable_object(&sum);
            tareador_end_task("jacobi_innermost_task");
        }
      }
    }

    ...
}
```

Figure 3: Code for the task decomposition for relax_jacobi function temporarily filtering the analysis of the sum variable.

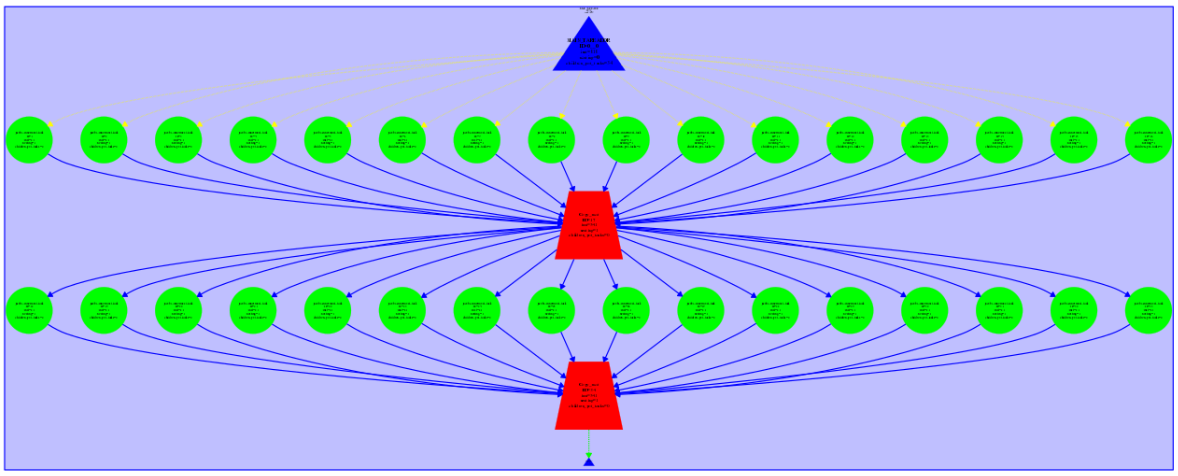


Figure 4: Task decomposition graph of the Jacobi solver temporarily filtering the analysis of the sum variable.

Now, we can see that there is any dependency between tasks of the most inner-loop. Hence, we are increasing the parallelism. We think that the *reduction(+:sum)* clause would be a good option to parallelise the code using *OpenMP* directives.

2.2 Gauss-Seidel Solver

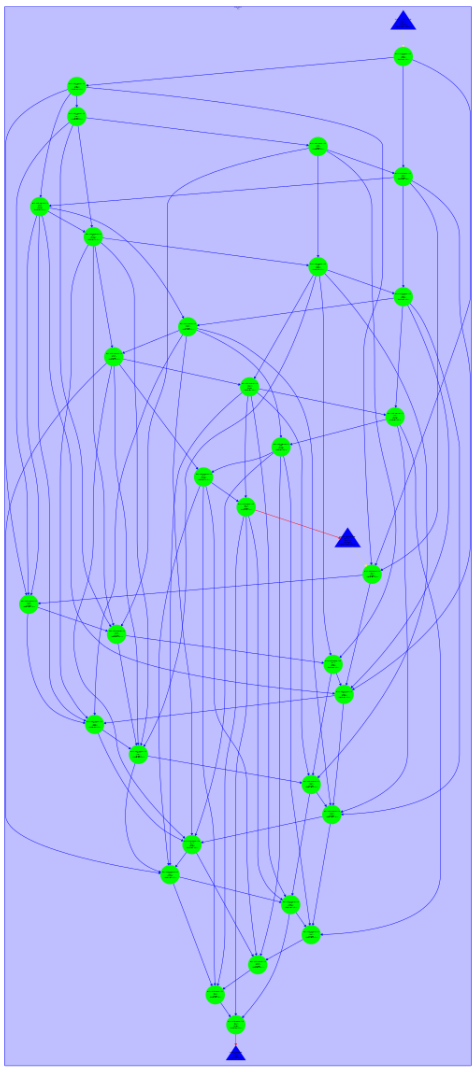
The Gauss-Seidel Solver does no longer use an auxiliar matrix. It writes in the position of the matrix where it reads the values. As in the previous solver, it takes the values of the element on its top and bottom and on its left and right, and does some arithmetic operations.

The modified code can be found in *gauss-seidel-tareador.c* file in the codes directory.

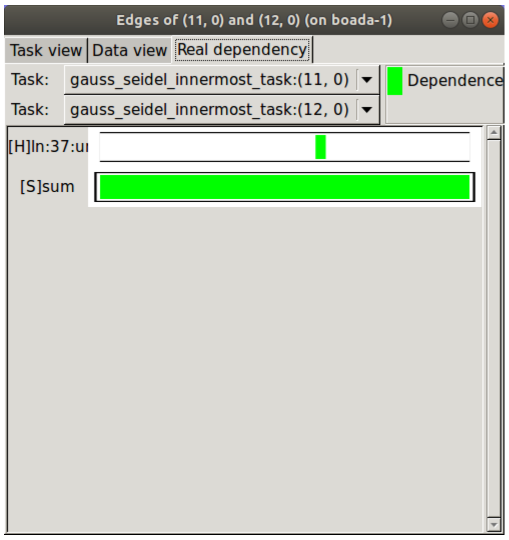
```c
double relax_gauss (double *u, unsigned sizex, unsigned sizey)
{
    double unew, diff, sum=0.0;

    int howmany=1;
    for (int blockid = 0; blockid < howmany; ++blockid) {
      int i_start = lowerb(blockid, howmany, sizex);
      int i_end = upperb(blockid, howmany, sizex);
      for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
        for (int j=1; j<= sizey-2; j++) {
            tareador_start_task("gauss_seidel_innermost_task");
            unew= 0.25 * ( u[ i*sizey   + (j-1) ]+  // left
                           u[ i*sizey   + (j+1) ]+  // right
                           u[ (i-1)*sizey  + j     ]+  // top
                           u[ (i+1)*sizey  + j     ]); // bottom
            diff = unew - u[i*sizey+ j];

            sum += diff * diff;

            u[i*sizey+j]=unew;
            tareador_end_task("gauss_seidel_innermost_task");
        }
      }
    }

    return sum;
}
```

Figure 5: Code for the task decomposition for relax_gauss function.

The task decomposition graph is shown in Figure 6a. We can observe that it has also some dependencies. Using the same procedure than in the previous section, we got the Real dependencies for this new solver (Figure 6b).



(a)



Edges of (11, 0) and (12, 0) (on boada-1)

Task view | Data view | Real dependency

Task: gauss_seidel_innermost_task:(11, 0)

Task: gauss_seidel_innermost_task:(12, 0)

🟩 Dependence

[H]ln:37:u

[S]sum

(b)

Figure 6: (a) Task decomposition graph for the Gauss-Seidel solver, (b) Real data dependences in the Gauss-Seidel solver.

We can see that this solver has two real dependencies: variable **sum** and some **positions of the matrix**. In this section, we will only show the TDG when disabling only the sum variable. However, the TDG of both variables disables can be found in the Annex section 4.1.

This new version of the code can be found in *gauss-seidel-disable-sum.c*, in the codes directory.

```
double relax_gauss (double *u, unsigned sizex, unsigned sizey)
{
    ...
    for (int blockid = 0; blockid < howmany; ++blockid) {
      ...
      for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
        for (int j=1; j<= sizey-2; j++) {
            tareador_start_task("gauss_seidel_innermost_task");
            ...
            tareador_disable_object(&sum);
            sum += diff * diff;
            tareador_enable_object(&sum);
            ...
            tareador_end_task("gauss_seidel_innermost_task");
        }
      }
    }
    ...
}
```

Figure 7: Code for the task decomposition for relax_gauss function temporarily filtering only the analysis of the sum variable.

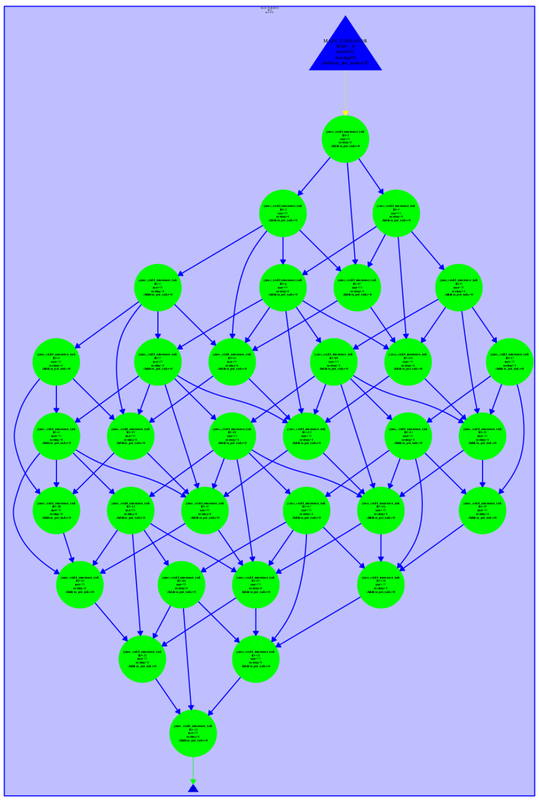


Figure 8: Task decomposition graph of the Gauss-Seidel solver temporarily filtering only the analysis of the sum variable.

Disabling temporarily the variable sum we see that we increase the parallelism.

Again, we think that the *reduction(+:sum)* clause would be good to porallelise the code using *OpenMP* directives. Moreover, we can use the *ordered* clause in order to avoid data races and respect the real dependencies we have seen (apply the doacross technique).

3 Parallelization of Jacobi with OpenMP parallel

In this part of the laboratory, we had to understand the code that resolves the problem using the Jacobi algoithm, and afterwards parallelize it, in order to increase the speed of the execution.

3.1 Understand the code

The Jacobi algorithm has some functions created in order to facilitate the readability of the code. This functions are:

```
#define lowerb(id, p, n) ( id * (n/p) + (id < (n%p) ? id : n%p) )
#define numElem(id, p, n) ( (n/p) + (id < (n%p)) )
#define upperb(id, p, n) ( lowerb(id, p, n) + numElem(id, p, n) - 1 )
#define min(a, b) ( (a < b) ? a : b )
#define max(a, b) ( (a > b) ? a : b )
```

Figure 9: Given functions

The function lowerb and upperb returns the first and the last index of a vector partitons, when it is given an id (the partition number), p (the number of partitions) and n (the size of the vector).

The function numElem that returns the number of elements that are in a partition of a vector. Having the same entries as before.

And finally min and max functions that returns the minimum and maximum number between two respectively.

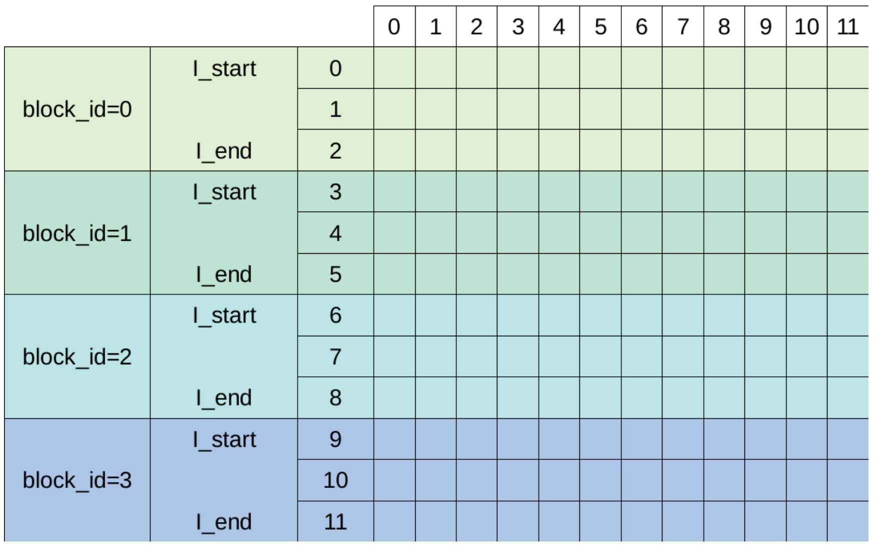


Figure 10: Geometric data decomposition.

In the geometric data decomposition we can see how the block_id is distributed in a matrix (12 x 12). Assuming that in the code is using a vector to represent the matrix, the size of the vector would be 144.

3.2 Parallelization of Jacobi

In this part we had to parallelize the code considering that we want to create 4 blocks, without using *#pragma omp parallel for* clause in the code. To do that we have modified the relax_jacobi function:

```
double relax_jacobi (double *u, double *utmp, unsigned sizex, unsigned sizey)
{
    double diff, sum=0.0;

    int howmany = 4;
    #pragma omp parallel reduction(+: sum) private(diff)
    {
        int blockid = omp_get_thread_num();
        int i_start = lowerb(blockid, howmany, sizex);
        int i_end = upperb(blockid, howmany, sizex);
        for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
            for (int j=1; j<= sizey-2; j++) {
                utmp[i*sizey+j]= 0.25 * ( u[ i*sizey + (j-1) ]+ // left
                                         u[ i*sizey + (j+1) ]+ // right
                                         u[ (i-1)*sizey + j ]+ // top
                                         u[ (i+1)*sizey + j ]); // bottom
                diff = utmp[i*sizey+j] - u[i*sizey + j];
                sum += diff * diff;
            }
        }
    }

    return sum;
}
```

Figure 11: Code for parallellization of relax_jacobi function

As we can see we have divided the matrix in 4 parts. This will cause that, if we execute the code with 8 threads only the first 4 threads will have a valid blockid number, so the other 4 threads will recieve an out of bound *i_start* index. As we can see in the following image:

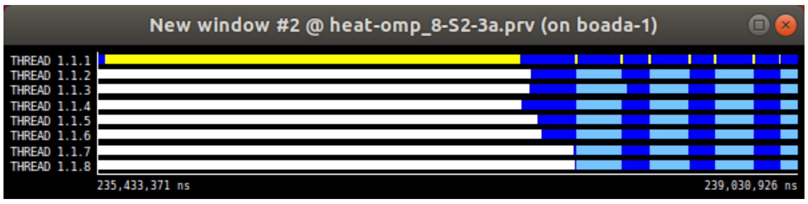


Figure 12: Jacobi Schedule.

In order to solve that we modified a little bit more the *relax_jacobi* function.

```
double relax_jacobi (double *u, double *utmp, unsigned sizex, unsigned sizey)
{
    double diff, sum=0.0;

    #pragma omp parallel reduction(+: sum) private(diff)
    {
        int howmany = omp_get_num_threads();
        ....
    }

    return sum;
}
```

Figure 13: Improved code of relax_jacobi function

Now we obtain a better parallelization in all the threads:

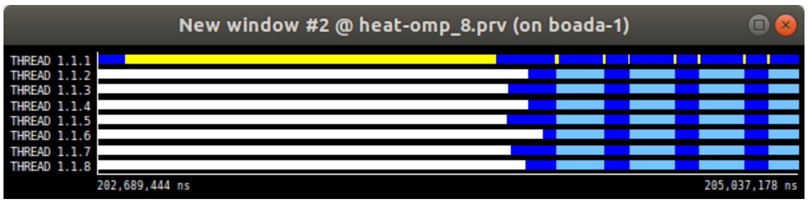


Figure 14: Improved Jacobi Schedule.

But this wasn't enough parallel. In that moment we realised that the function *copy_mat* also has a big impact in the code. SO we decide to parallellize it.

```
void copy_mat (double *u, double *v, unsigned sizex, unsigned sizey)
{
    #pragma omp parallel for collapse(2)
    for (int i=1; i<=sizex-2; i++)
        for (int j=1; j<=sizey-2; j++)
            v[ i*sizey+j ] = u[ i*sizey+j ];
}
```

Figure 15: Code for parallellization of copy_mat function

Now we obtained the best results regarding to parallelization. AS we can see in the following image:

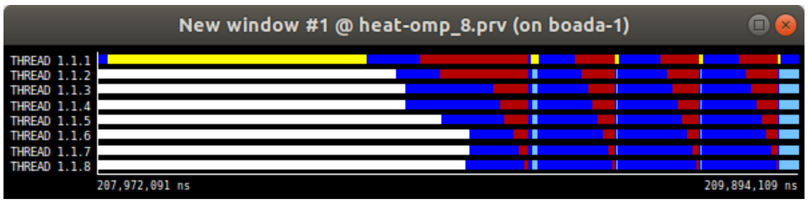


Figure 16: Improved Jacobi Schedule with copy_mat parallellization.

Finally in order to test the scalability of our program. We executed the script *submit-strong-omp.sh*, to obtain the following speed.up and scalability plots.

Figure 17: Jacobi scalability plots.

As we can see the performance increases together with the number of threads at first. But when we have a large number of threads the results shows a bit stagnation of this improvement.

4 Annex

4.1 Task dependency graph when also disabling temporarily some positions of the matrix

In this section we will see the TDG created when disabling all the variables that create some kind of dependency in the Gauss-Seidel solver. The code can be found in *gauss-seidel-disable-sum-and-matrix-positions.c*, inside the codes folder.

```
double relax_gauss (double *u, unsigned sizex, unsigned sizey)
{
    ...
    for (int blockid = 0; blockid < howmany; ++blockid) {
      ...
      for (int i=max(1, i_start); i<= min(sizex-2, i_end); i++) {
        for (int j=1; j<= sizey-2; j++) {
            tareador_start_task("gauss_seidel_innermost_task");

            tareador_disable_object(&u[ i*sizey    + (j-1) ]); // left
            tareador_disable_object(&u[ (i-1)*sizey    + j      ]); //top
            unew= 0.25 * ( u[ i*sizey  + (j-1) ]+  // left
                           u[ i*sizey  + (j+1) ]+  // right
                           u[ (i-1)*sizey  + j      ]+  // top
                           u[ (i+1)*sizey  + j      ]); // bottom
            diff = unew - u[i*sizey+ j];
            tareador_enable_object(&u[ i*sizey + (j-1) ]);
            tareador_enable_object(&u[ (i-1)*sizey + j      ]);

            tareador_disable_object(&sum);
            sum += diff * diff;
            tareador_enable_object(&sum);

            ...
            tareador_end_task("gauss_seidel_innermost_task");
        }
      }
    }
    ...
}
```

Figure 18: Code for the task decomposition for relax_gauss function temporarily filtering the analysis of the sum variable and some positions of the matrix.

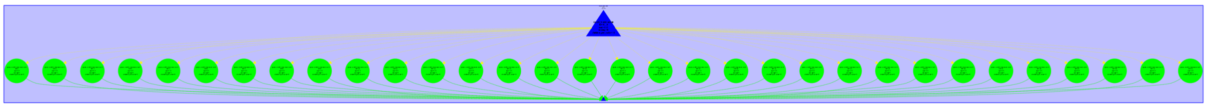


Figure 19: Task decomposition graph of the Gauss-Seidel solver temporarily filtering the analysis of the sum variable and some positions of the matrix.